GhostMinion: A Strictness-Ordered Cache System for Spectre Mitigation

GhostMinion: A Strictness-Ordered Cache System for Spectre Mitigation

Sam Ainsworth
University of Edinburgh
sam.ainsworth@ed.ac.uk

Abstract

Out-of-order speculation, a technique ubiquitous since the early 1990s, remains a fundamental security flaw. Via attacks such as Spectre and Meltdown, an attacker can trick a victim, in an otherwise entirely correct program, into leaking its secrets through the effects of misspeculated execution, in a way that is entirely invisible to the programmer's model. This has serious implications for application sandboxing and inter-process communication.

Designing efficient mitigations that preserve the performance of out-of-order execution has been a challenge. The speculation-hiding techniques in the literature have been shown to not close such channels comprehensively, allowing adversaries to redesign attacks. Strong, precise guarantees are necessary, but mitigations must achieve high performance to be adopted. We present Strictness Ordering, a new constraint system that shows how we can comprehensively eliminate transient side channel attacks, while still allowing complex speculation and data forwarding between speculative instructions. We then present GhostMinion, a cache modification built using a variety of new techniques designed to provide Strictness Order at only 2.5% overhead.

1 Introduction

Speculative side-channel attacks such as Spectre [18] and Meltdown [21] have severely damaged modern systems' security: even functionally correct programs can be compromised through the leakage of secrets via incorrectly predicted program behaviour. Still, the side channels that cause this leakage, for example caches, are desirable in many cases; the speed of modern microarchitectures is only possible because of structures that leak soft state. Out-of-order speculation, which allows attackers to execute complex speculative code, is mandatory for high instruction-level parallelism.

Many different mitigation techniques, with differing threat models and security properties, have been proposed. Often the tradeoffs are unclear, as the guarantees necessary, and provided, are typically opaque. This has resulted in subtle bugs in many proposed countermeasures. Since precise models of the behaviour guaranteed by mitigations are typically unavailable, sophisticated attacks to circumvent restrictions, such as SpectreRewind [10] and Speculative Interference [5] have appeared to fill in the gaps. We now know that cleaning up the effects of speculation after it has occurred is insufficient to prevent microarchitectural speculation attacks. This means that speculation-hiding techniques in the literature [1, 27, 37, 38] leak, through inability to correctly handle concurrent execution and incomplete coverage of microarchitectural structures. Examples include but are not limited to allowing structural hazards in non-pipelined divider units to be caused by otherwise "invisible" speculative loads [10], and can cause comprehensive breaches, and leave long-lasting traces in otherwise protected cache systems [5]. Speculation-restricting techniques [20,29,36,40], by comparison, prevent many executions that are completely valid, with significant performance loss for memory-intensive workloads.

We present Strictness Ordering, a permissive constraint system that allows information (and side-channel) flow in any situation where speculative operations that do not commit can never leak to those that do. This is complicated by the fact those instructions may execute concurrently [5, 10], but can be expressed as a simple property (definition 1). This allows significantly more freedom than only allowing instructions to emit timing side-effects once non-speculative [20, 29, 36, 40], because whether an instruction commits or not is heavily tied with the fates of other concurrently executing instructions. For example, instructions in a pipeline only commit when all previous instructions in program order do so, and so those previous instructions can freely transmit data to, and take resources from, concurrently executing future instructions.

We show how to build Strictness Ordering into microarchitecture with GhostMinion, a cache system and core modification designed to eliminate transient execution attacks including Spectre [18], and associated derivatives including SpectreRewind [10] and Speculative Interference [5]. Overheads from the cache modification are just 2.5% on SPEC CPU2006, 0.6% on SPECspeed 2017, and negligible on Parsec, and we argue that fixing other Strictness-Order issues within the system are unlikely to have significant overhead. GhostMinion defeats transient execution attacks more comprehensively than previous speculation-hiding defences [1, 11, 16, 27, 28, 37, 38], and without fully associative overprovisioned structures [16, 38].

Our contributions are as follows:

- We define a timing model, Strictness Order, that if obeyed can eliminate all forms of transient execution attack, including backwards-in-time and contention attacks [1, 5, 10], while still being highly permissive to innocuous forms of information flow and out-of-order execution.
- We define Temporal Order as a simple overapproximation of Strictness Order.
- We develop GhostMinion, a cache system that uses the new techniques of TimeGuarding, Lapfrogging and

Timeleaping to enforce Temporal Order, and thus Strictness Order, in the cache hierarchy.

- We implement or describe the extensions that would be necessary to achieve Temporal Order and/or Strictness Order for the entire system, for hypothetical attacks that violate Strictness Order (via contention or soft state) anywhere within a processor, and argue that the techniques demonstrated by GhostMinion can be used elsewhere and the additional overheads are likely to be neglgible.
- We evaluate GhostMinion in gem5, showing overheads of only 2.5% geomean for SPEC CPU2006, 0% for Parsec, and 0.6% for SPECspeed 2017.

1.1 Threat Model

We assume the existence of an attacker, trying to leak secret data through transient (misspeculated) execution. They will succeed if they can change the execution time of *any* committed instruction at any point before, during or after the misspeculation, via a transient access of the secret they are trying to leak. This measurement includes but is not limited to the execution time of instructions that directly measure time, such as `rdtsc`. The attack may be via attacks such as Spectre [18], where no explicit exceptional behaviour is triggered by the misspeculation, or attacks such as Meltdown [21] or Foreshadow [9] where exceptional behaviour is incorrectly temporarily ignored by the speculative execution, which can be fixed more directly. When we refer specifically to Spectre, we do so because it requires more all-encompassing strategies for defence, and without loss of generality.

We do not necessarily require the attacker to successfully bring in or evict data from any observable cache past the point of misspeculation. This is because recent attacks [10] are able to measure secrets just from the effects on concurrently executing instructions, and such contention effects are able to be converted into long-lasting state changes [5].

We assume the attacker is able to run arbitrary code, but cannot load secret data directly; e.g., they are within a sandbox either in userspace or the kernel, sharing an address space with the secret but with runtime checks preventing access. The attacker could also reside in another process [1], or on another system entirely [31].

2 Background

2.1 Spectre

Spectre [18] uses transient misspeculation of a processor, such as from a mistrained branch predictor or branch target buffer, to accesse information that the correct execution of the program would not. This can be used by an attacker to trick a victim into loading secret data, for example by ignoring safety checks on inputs or within sandboxes. Though the execution is rolled back, it may leave traces within soft-state structures such as caches, which can be measured using timing side channels to reveal the secret.

In the original Spectre proof-of-concept [18], the attacker collects secret data by using it to then trigger other transient loads into the cache based on using the secret value as an index. This is followed by measuring which cache lines have been brought into the cache once correct execution restarts, in an evict-and-time attack, or evicted, via a prime-and-probe attack. This gave rise to defences that cleared up misspeculation effects after discovery [1, 11, 27, 28, 37].

2.2 Backwards-in-Time Attacks

Post-misspeculation rollback is insufficient against attacks that rely on changing the timing of committed instructions *before* the misspeculation, such as SpectreRewind [10]. Out-of-order processors execute many instructions in parallel. Though the frontend fetches instructions in sequence, and the backend retires them in sequence, in the middle, operations that do not commit (and are misspeculated) can occur before those that do, which are earlier in program order. SpectreRewind [10] uses misspeculation to change the timing of these logically earlier (but executed later) instructions, by causing structural-hazard contention in non-pipelined divider units. Speculative Interference [5] takes this one step further: it notes that these timing effects on logically earlier instructions can actually change the order in which instructions execute, causing different state to be stored in the cache system, and thus leaving a long-lasting trace of behaviour that can later be picked up by an attacker. The effect is not limited to non-pipelined functional units; equivalent contention can be caused by transient instructions bringing data into, or evicting data from, the cache for earlier concurrently executing instructions to use [1]. Clearing state changes on the discovery of misspeculation [1, 11, 27, 28, 37] is insufficient: side channels can travel *backwards in time* to cause equivalent damage.

3 Strictness Ordering

Speculative-execution attacks cause measurable behaviour that can be exploited to retrieve secrets. This is achieved by causing transient instructions, which would never be executed without the presence of misspeculation, to affect the timing of other instructions that are otherwise correct. To complicate this, instructions within a pipeline are not cleanly split up in to "transient" and "non-speculative", where the latter can transmit to any instruction and the former cannot: instead, a pipeline is filled with instructions from progressive levels of speculation. During execution it is typically unknown how far down the path of instructions we will commit before a transient misspeculation, and instruction flush, will occur.

Still, we must not be overly restrictive, in capturing innocuous transmission between instructions that cannot leak transient execution to committed state. To directly capture the threat of instructions that do not commit affecting the timing of those that do, we define a new timing influence model that must be obeyed to prevent speculative timing-side-channel attacks (definition 1).

Definition 1 (Strictness Ordering)**.** *Let* x *and* y *denote executed instructions. Under Strictness Ordering,* y *can* strictly observe x *($x \overset{S}{\Rightarrow} y$), meaning* x *can impact the execution time (cycle of any pipeline stage) of* y*, iff*

$$commit(y) \rightarrow commit(x)$$

where $commit(i)$ is true iff instruction i is guaranteed to reach the end of the CPU pipeline without being squashed, or already has, and $\rightarrow$ is logical implication.

In other words, for y to observe any timing change from the execution of x, y must be an instruction that is always squashed whenever x is squashed, or x must be an instruction that will not be squashed (or has already committed). Intuitively, this can be viewed as preventing timing side-channel effects from flowing *backwards-in-time* from more speculative to less speculative instructions.

To prove this is sufficient to eliminate transient side-channel attacks, suppose x is an instruction that is part of a transient-execution gadget: it can observe secret data D that is inaccessible under correct execution, and thus never commits, so $\neg commit(x)$. Since x does not commit, neither will any instructions that depend on it, and so D may only be recovered via a (timing) side channel. If we assume the attack succeeds, some instruction y exists, such that $commit(y)$ holds, that has its timing impacted by the execution of x to measure the channel[a]. For that to have occured, $x \overset{S}{\Rightarrow} y$ holds by definition, and so $commit(y) \rightarrow commit(x)$. Since $commit(y)$ is true, $commit(x)$ must also hold. However, since x is transient, $\neg commit(x)$ also holds, and we have a contradiction: no transient x that can transmit data to a committed instruction via timing side channel exists.

[a]This may be a timer such as `rdtsc`, or be based on the commit time of an instruction with state, such as an incrementing counter [30].

There are a number of properties that follow:

Every instruction that has already committed, or is guaranteed to commit, can transmit to any other. Side-channel flow will always eventually be able to occur for valid instructions executing in program-order, as a fallback for more advanced reordering.

Within a thread and pipeline, for any two instructions a and b, either $a \overset{S}{\Rightarrow} b$, $b \overset{S}{\Rightarrow} a$, or both. Assume that, as is typical, an out-of-order processor issues instructions (speculatively) in order at the front-end, and that misspeculation is handled by restarting the pipeline at the last known correct instruction in program order. It follows that there is a total order on instructions that commit: for an instruction to commit, all earlier instructions issued in program order must also do so, thus at least one of $commit(a) \rightarrow commit(b)$ or $commit(b) \rightarrow commit(a)$ holds. This allows us to be more lenient than waiting for instructions to become non-speculative, because it always allows older instructions to immediately transmit to newer instructions in program order, even while speculative.

The relation is transitive. Imagine that a transient instruction x affects the execution time of an instruction y that will commit. Even if the commit time of y is unchanged, and no directly visible side channel occurs, if the result forwards to another committed instruction z, which then commits earlier as a result, a side channel becomes visible. These transitive effects make it sufficient for Strictness Order to only maintain that the *commit time* of committed instructions to be unaffected rather than *execution time* per se, as a change in execution time of one instruction can be converted to a change in commit time of another if it has the potential to become visible. Still, to avoid having to consider this transitive impact, it is simpler to design hardware that preserves guarantees over all *execution time*: that is, at no pipeline stage should the effects of a transitive instruction alter the timing of an instruction that may later be committed.

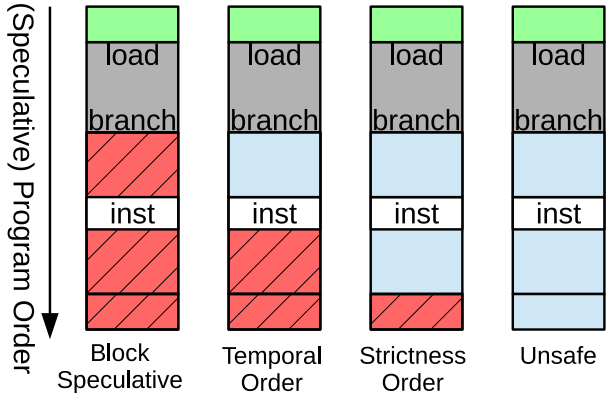


Figure 1: An example of ordering restrictions. Instructions in green are non-speculative, as are those in grey, which are held up by the load's cache miss. Instructions following the grey branch are speculative. Instructions in red (with hatching) cannot affect the timing of the instruction in white before becoming non-speculative, whereas instructions in blue may do so: under Strictness Order, any load before the blue branch can forward data to, or evict data seen by, the instruction in white, or contend for the same resource. By contrast, preventing speculation (Block Speculative) disallows many valid flows, whereas a baseline Unsafe system allows transient execution attacks, by making the effects of operations that do not commit, and can access data disallowed by correct execution, visible to other instructions.

Between threads, no such program ordering holds. For two instructions in two threads x_{T1} and y_{T2}, $x_{T1} \overset{S}{\not\Rightarrow} y_{T2}$ and $y_{T2} \overset{S}{\not\Rightarrow} x_{T1}$ may both hold. Indeed, there is unlikely to be any reason for an ordering outside a guarantee of commit: timing information from y_{T2} can typically only be transmitted to $T1$ once the y_{T2} instruction becomes non-speculative.

The result is a preorder relation[1], demonstrated in fig. 1, which allows significantly freer execution than many previous techniques [20, 29, 36, 40], without affecting security.

Outside of its basic definition, for the Strictness Order property to be enforceable, two other properties are assumed.

First, speculative execution can only be measured by timing impact on at least one valid instruction. Outside of access to advanced performance counters, this is typically true. Second, if $y \overset{S}{\not\Rightarrow} x$, then as well as x's execution time, y must also not be able to affect *whether* x commits. A speculative tech-

[1]Strictness Order is named after, and closely related to, strictness analysis [24], an optimisation in software to transform call-by-need into call-by-value. A function f is strict in an input x if the function fails to terminate whenever the input fails to terminate. In the same way, an instruction y can strictly observe an instruction x under Strictness Order if whenever x fails to commit, y fails to commit.

nique, that assumed $y \overset{S}{\Rightarrow} x$ and rolled back and re-executed x if incorrect, would affect the timing of the program based on the behaviour of y, an instruction that was never committed.

Strictness Ordering is sufficient to comprehensively prevent any transient execution attack, regardless of whether the effects are timed at any point after correct execution has restarted [5, 18, 21] or whether the observed effects are entirely concurrent contention and do not persist past the return to correct execution [5]. This is because no misspeculated operation can ever affect the execution time of any committed instruction at any point.

> **Definition 2** (Temporal Ordering)**.** *Let* x *and* y *denote executed instructions. Under Temporal Ordering,* y temporally succeeds x *($x \overset{T}{\Rightarrow} y$), meaning* x *can impact the execution time of* y*, iff*
>
> $$commit(x) \vee seq(x, y)$$
>
> *where $commit(i)$ is true iff instruction i will reach the end of the CPU pipeline without being squashed, or already has, $seq(x, y)$ indicates that* x *occurs before* y *in the program's instruction sequence, and $\vee$ is logical disjunction.*

A simple overapproximation of Strictness Order is Temporal Order (definition 2). This considers each new instruction as more speculative than the last. This is more lenient than in-order execution, or blocking all speculative computation (Block Speculative in fig. 1), as it only prevents *information flow* (side channels or otherwise) in reverse. Execution can otherwise still occur out-of-order. For typical out-of-order systems, which re-execute instructions past misspeculation, a Temporal-Order microarchitecture obeys Strictness Order: since there is a total order on instructions that commit within a thread, $seq(x, y) \rightarrow (commit(y) \rightarrow commit(x))$, and by logical tautology, $commit(x) \rightarrow (commit(y) \rightarrow commit(x))$. Instructions can be treated as speculative until they reach commit, and each new instruction can be treated as more speculative than the last. GhostMinion implements Temporal Order for simplicity, and because the overapproximation causes little performance loss even though it rejects some valid orderings. The difference is shown in fig. 1.

4 GhostMinion

Here we develop GhostMinion, an extension to the cache system and core to enforce Temporal Ordering (definition 2), and thus Strictness Ordering (definition 1) as a result. A summary is shown in fig. 2: we cache the results of speculative memory accesses in a compartment of the L1 cache, labelled a GhostMinion. These are only allowed to impact state within the rest of the cache system (the non-speculative cache hierarchy) once an instruction becomes non-speculative. Within the GhostMinion, instructions may be forbidden from seeing eviction or insertion of others, depending on their Strictness Order (figs. 3 and 4). The enforcement of this is guaranteed by *TimeGuarding*, which directly translates Temporal Order into hardware. Transient contention in the rest of the system, such as the miss-status handling registers (MSHRs) or non-pipelined instructions, is also subject to TimeGuarding, or when more appropriate *leapfrogging* (fig. 5), which allows stealing of resources, followed by replay of the offending operation, to prevent breaking Strictness Order under greedy out-of-order resource allocation.

4.1 How to Achieve Strictness Ordering

Suppose we have two instructions, x and y, where x is a misspeculated instruction transmitting via side channel to y, which has, or will be, committed. There are three cases: y commits before x enters the pipeline, y is issued after x is flushed, or both are in the pipeline concurrently.

The first case is trivial. If y commits before x enters the pipeline, then no timing information from x could ever reach y. The second is also relatively simple: we must clear any state changes that were caused by misspeculation before restarting correct execution. In order to not affect the timing of future committed instructions, thus violating Strictness Order, this rollback to a misspeculation-free state must be timing-invariant to the content, or *volume* of content, that needs to be cleaned up. This means that allowing misspeculation to propagate through a cache system, followed by (serially) undoing [27, 37] the results would violate Strictness Order, resulting in a side channel. Instead, we must make this rollback time-invariant [1] and efficient, by limiting speculative changes to a small region of SRAM (the GhostMinion).

The final case, where x and y execute concurrently, is that exploited by techniques such as SpectreRewind [10] and Speculative Interference [5]. Since both are in the pipeline simultaneously, and y will be committed but x will not, we know that y is before x in (speculative) program order, and that $y \overset{S}{\Rightarrow} x$ but $x \overset{S}{\not\Rightarrow} y$. Still, y may be delayed from being a candidate for execution, and so x may execute first. To enforce Strictness Order, x's execution must not be visible by timing y. Not only must we keep speculative resources confined to a GhostMinion: we must also prevent any *backwards-in-time* channels before misspeculation is detected. Under Temporal Order, which treats instructions as a sequence, older instructions in program order must not be able to read data brought in by logically newer instructions, and so must observe a cache miss in the GhostMinion (fig. 4a). Newer instructions also cannot evict the data of an older instruction, and so a fill must fail if there are no storage options left within a GhostMinion cache set that do not violate this constraint (fig. 4b). Resource contention must also be hidden, and so older instructions can steal contended resources taken by newer instructions, which must then repeat (fig. 5).

4.2 Overview

GhostMinions are placed next to the L1, and accessed in parallel with the L1. They buffer speculative accesses from the rest of the memory hierarchy, which cannot be altered by speculative evictions or fills. The *non-speculative* cache hierarchy may follow any inclusion or exclusion policy, but must be non-inclusive non-exclusive with respect to the GhostMinion to avoid information leakage.

The non-speculative cache hierarchy never sees state changes from speculative instructions, and so no rollback is required for them on misspeculation. Only the GhostMinion is wiped on misspeculation, and only it needs restrictions on its data forwarding (fig. 4). Changes in the non-speculative

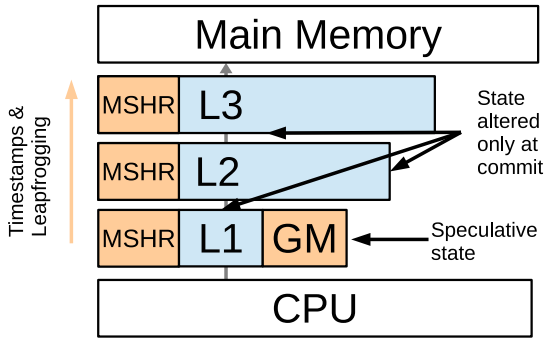


Figure 2: GhostMinion attaches to each L1 cache, and is designed to cache speculative loads, with *TimeGuarding* to make this data (and any eviction) invisible under concurrent misspeculation. To hide contention, timestamp metadata propagates into the MSHRs at each cache level, allowing loads to be cancelled and replaced (*leapfrogging*).

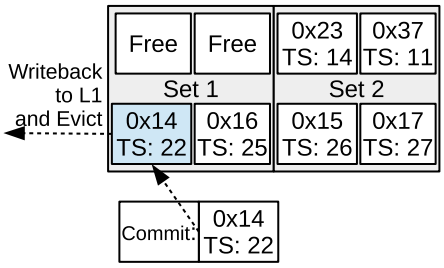


Figure 3: On commit of a load, the GhostMinion is checked for any cached line that matches, and can be validly read in Strictness Order by the committed instruction based on its timestamp (TS). If found, it is written to the L1, and evicted from the GhostMinion, leaving a "free slot" for future speculative loads to fill without evicting non-speculative data.

cache hierarchy can only occur through non-speculative memory accesses (such as the head of the reorder buffer), through a committed instruction moving data out of the GhostMinion (fig. 3), or through the actions of a prefetcher. Still, GhostMinions share the L1's miss status handling registers (MSHRs), and use the L2 and L3's MSHRs, which though they are transient, can cause contention-based timing flow without care. To avoid this, we propagate Temporal Order throughout the MSHR hierarchy, in particular using leapfrogging (fig. 5) to steal resources back when the existence of a partially complete memory access would leak. GhostMinions cannot cause deadlock or livelock, as all valid instructions become non-speculative, and thus globally visible, once they reach the head of the reorder buffer.

GhostMinions are wiped on misspeculation, in a timing-invariant way, to clear all timestamps above the point of misspeculation[2]. This is done in a single cycle using parallel registers [1], separate from the L1 cache, which indicate validity and timestamp. GhostMinions do not hold dirty data, and so no writeback is required on this invalidate operation. In addition, to prevent Temporal-Order violation from concurrent execution, constraints on forwarding and evicting data in a GhostMinion must hold: these are explored below.

4.3 Free-Slotting

A speculative overwrite (by an instruction x), within the GhostMinion, of data brought in by an instruction z that has committed, is forbidden, as this would result in a backwards-in-time channel by evicting the non-speculative data. Precisely, a speculative instruction y, where $z \overset{T}{\Rightarrow} y \overset{T}{\Rightarrow} x$, but $x \overset{T}{\not\Rightarrow} y$, may then try to read the original, now-evicted cache line from z. What follows is that, in order to bring data into a GhostMinion, we can only overwrite free slots or data that is *more speculative* than the current load. To avoid resource starvation, when we write a cache line into the non-speculative L1 on commit, we must also evict it from the GhostMinion to create a *free slot* for speculative memory accesses (fig. 3). If the cache system guarantees either exclusion or inclusion for the non-speculative hierarchy, then the line is kept in the GhostMinion until the data can be moved to the L1 while preserving this invariant, to service requests while exclusion or inclusion are enforced in the hierarchy.

Free-slotting is why GhostMinions are placed next to L1 caches, rather than closer to the CPU as in techniques such as MuonTrap [1]. Since a GhostMinion does not store committed data, it no longer makes sense to use a GhostMinion as a fast L0 cache for frequent lookups: instead, it should be accessed in parallel with the L1. This writeback-on-commit is the only way data can escape to the outside world: otherwise, it is thrown away, to avoid affecting other caches[3].

4.4 TimeGuarding

To track Temporal Order, each issued instruction is assigned a timestamp. This is attached to speculative memory accesses of the GhostMinion; a speculative load is only allowed to *read* a value from the GhostMinion if its timestamp is *greater or equal* than the value in the GhostMinion (fig. 4a).

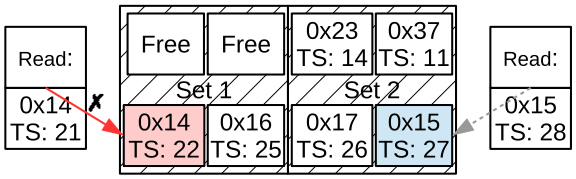
A cache *fill* in the GhostMinion is only allowed if *a)* the block is free, or *b)* the new request has a *lower* timestamp than the block it wishes to evict (fig. 4b)[4]. Whether a suitable space is available or not is dictated by the value of a given address, assuming the structure is not fully associative, and the number of sets available. If no sets can validly be used within a given line without violating TimeGuarding, the load is returned to the CPU without being written to the GhostMinion: it will thus not be available to write back to the L1 on commit.

Logically and behaviourally, the TimeGuard can be considered a timestamp increasing to infinity. Implementation-

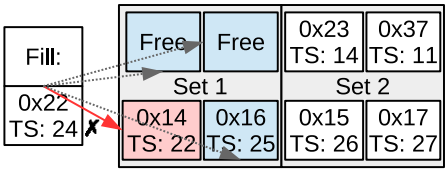
[2]Unlike in MuonTrap [1], we do not clear *all* data from the GhostMinion, as this could violate Temporal Order if the discovered misspeculation is *itself* speculative (e.g. because a misspeculated branch C relies on a yet-to-be-resolved branch B).

[3]Nevertheless, the state of the non-speculative cache hierarchy can validly be different from a sequential execution, but only in ways that do not violate Temporal Order. For example, if instruction y at timestamp 43 fills the GhostMinion, then x at timestamp 42 fills to the same slot, which it sees as free to preserve Timestamp Order. This means that y, never fills to the L1 despite both being valid. This is permitted because, while x cannot see the out-of-order effects of y, y is validly permitted see the out-of-order effects of x. Thus, maintaining equivalent state to an in-order execution is not necessary to maintain Temporal Order or Strictness Order.

[4]Any policy that does not reveal the difference between *a)* a "free slot" and *b)* a location taken by a block at a higher timestamp, to an instruction at a lower timestamp, is valid; we choose to take free slots if available, and evict the highest-timestamped available location in a set otherwise. This is because only the highest-timestamped instruction is aware that the GhostMinion is in fact full.



(a) Reads only succeed if the timestamp of a cache line is less than or equal to the timestamp of the instruction. In this example, if the load of *0x14* at timestamp 22 did not commit, but the same load at timestamp 21 committed and observed 22's cache line, this would break Strictness Order. For the load of *0x15*, the timestamp 28 is higher than 27; the former cannot commit if the latter fails, so the channel is permitted.



(b) Filling the GhostMinion is the opposite: it can only overwrite data with a greater-than or equal timestamp (or a free slot).

Figure 4: To enforce Strictness Order, GhostMinion puts TimeGuarding restrictions on which data can be read or overwritten, to make the existence of concurrent misspeculation invisible. This prevents any backwards information flow, from existence or non-existence of data, and works regardless of the associativity of the GhostMinion.

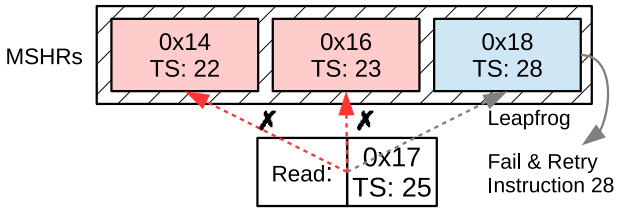


Figure 5: An example of *leapfrogging*. Here, no miss status handling registers (MSHRs) in the cache are free. The read request we are attempting has a timestamp of 25, and so cannot affect the execution of the MSHR requests at timestamps 22 and 23. The timestamp 28 must be invisible to our current request: the new read, at timestamp 25, replaces the one at timestamp 28, which fails and must reattempt later.

wise, the maximum timestamp is sized to be twice the number of reorder-buffer entries, as a sliding window[5].

4.5 Leapfrogging

GhostMinion allocates and schedules resources to instructions greedily. A speculative instruction y may be allocated the last free MSHR in the cache, when an instruction x, before y in program order, becomes available to execute. If y then delayed the execution of x, this would be a violation of Temporal Order (and thus Strictness Order), and so to fix this, x must be able to steal y's MSHR from underneath it, in a process called *leapfrogging* (fig. 5). This causes the load for y to lose its MSHR resource; the core treats it as though the MSHRs are full, and so the load must be reattempted once new resources are available, to hide its execution from x. To implement this, a thread's load requests' timestamps are propagated up the entire MSHR hierarchy: a load can fail due to leapfrogs in any MSHR up to main memory.

A special case of leapfrogging, which we call *timeleaping*, is when instructions x and y, such that $x \overset{T}{\Rightarrow} y$ but $y \overset{T}{\not\Rightarrow} x$, execute concurrently, with y issuing first, and both attempt to bring in the *same* cache line. Here, they can both be attached to the same MSHR, and so y need not be cancelled and replayed[6]. To hide the timing effect, the load is still leapfrogged: it is restarted at each cache. As with standard leapfrogging, this may cause *cascading leapfrogs*, in this case of the same request, in multiple different cache levels. This achieves the same timing response as if only x executed the load, which since $x \overset{T}{\Rightarrow} y$ may validly forward the result to y.

4.6 Coherence

Timing data can also leak through influencing coherence protocol states [34]. Between threads, no Strictness Order exists in general (section 3). This means that coherence states, either in non-speculative caches or in other GhostMinions, must not be altered until an instruction commits.

GhostMinion uses a technique taken from MuonTrap [1], which is sufficient to avoid altering non-speculative coherence state, and avoid observing timing effects from speculative state. A block within a GhostMinion can only be in Shared or Invalid[7]. A *Shared* block can only exist in a GhostMinion provided no *Exclusive/Modified* blocks sit in non-local parts of the cache hierarchy; that is, caches not along a direct path from the CPU to Main Memory, for example in a private (non-speculative) cache of another core. Loads that would violate this wait until non-speculative, filling directly into the L1, before they can gain a coherent copy.

Equivalently, if a line exists in exclusive or modified in a localised part of the non-speculative cache (for example, a private L2 or L1), a valid copy cannot exist, in a GhostMinion or in a non-speculative cache, down any other part of the hierarchy. This means any upgrade (for example, to perform a store) need only invalidate GhostMinions on paths *closer to a CPU* than such a copy, in a method timing-insensitive to the speculative contents of a GhostMinion [1].

Outside the coherence protocol itself, GhostMinion can forward a speculative non-coherent copy of a value, in Exclusive or Modified in a non-local part of the cache hierarchy, to the GhostMinion. On an attempt to commit an

[5] Since there will only ever be N maximum instructions in the reorder buffer at once, allocated in timestamp order, each individual instruction can only execute concurrently with $2N$ instructions. It is sufficient to allocate timestamps in-order with a wrap-around at $2N$, and gives the constraint that an instruction at timestamp t can only overwrite a cache line at timestamp $t..((t+N) \mod (2N))$, and only read all others.

[6] If x literally takes the same MSHR, and thus prioritises the taking of a *particular* MSHR based on whether x and y match, then leapfrogging should otherwise occur on the highest timestamped MSHR to avoid breaking Temporal Order. If we instead used a random eviction policy of MSHRs for any timestamp above x's, an instruction z using another MSHR, which believed there was still free space, could learn that the addresses of x and y match even if $x \overset{T}{\Rightarrow} z \overset{T}{\Rightarrow} y$ and $y \overset{T}{\not\Rightarrow} z \overset{T}{\not\Rightarrow} x$, if it were therefore never leapfrogged.

[7] The ReadExclusive memory request necessary for stores can only occur once the store becomes non-speculative [1]. Stores already typically occur at commit, since they are challenging to roll back, and do not sit on critical paths unlike loads, due to the store buffering permissible in conventional TSO and relaxed consistency models.

instruction using this non-coherent copy, the load is replayed non-speculatively, and the instruction itself re-executed if the value has subsequently changed. This technique is taken from InvisiSpec [38], which uses it for all speculative loads. If the load originally received a valid shared copy, rather than a value-predicted version, and can thus commit without this check, and no shared copy existed in any non-local part of the non-speculative hierarchy, the L1 cache may then issue a decoupled upgrade [1] to move the data into exclusive.

4.7 Prefetching

To avoid information leakage, prefetchers in non-speculative parts of the cache hierarchy can only be trained on non-speculative input. To achieve this, speculative memory accesses are tagged with the cache level they were brought in from. On commit, not only is the value (if relevant) brought in to the GhostMinion, committed to the L1, a prefetcher notification is also sent to the level it was brought in from (and any levels closer to the CPU), provided it has a prefetcher at that level. Again, this technique is inspired by MuonTrap [1].

Prefetchers can only be trained using data from speculative streams if they prefetch into the GhostMinion. For prefetchers close to the CPU, such as fetch-directed instruction prefetching [26], this may be desirable. These prefetches are timestamped to the value of the last instruction used as input to the prefetcher: only instructions with equal or higher timestamp can observe the prefetches.

4.8 Instruction GhostMinion

The instruction cache can also be affected by transient execution attacks. These can be caused in two ways. Branches can be speculatively resolved not only by lookup in the predictor and branch target buffer, but also as a result of computation (for example, a branch or branch target conditional on the value of a secret). Alternatively, secrets can generate a contention effect on instruction-fetch progress.

Because instruction fetch is executed in (speculative) program order, protection is easier than for the data hierarchy, since no instruction reordering can occur, so for a pipeline's concurrently executing instructions, Temporal Order cannot be broken via passing information *backwards-in-time*. We need only worry about leaking information from the end of misspeculation to the start of re-execution. Still, we must be able to return to a cache state unaffected by wrong speculation, and in a timing-invariant way that does not reveal how much state was changed. This requires being able to wipe an instruction GhostMinion with a parallel invalidate operation, like for a data GhostMinion. Further, if $y \overset{\mathrm{T}}{\not\Rightarrow} x$, then y cannot evict x's cache line unless x has been committed. We handle this via TimeGuarding (section 4.4), and in practice, the instruction GhostMinion is similar to the data GhostMinion mentioned above, without coherence requirements.

4.9 Remaining Channels

Though for a system to truly be transient side-channel free, the entire system must obey Strictness Order, the techniques here are sufficient to prevent load forwarding, either forwards or backwards in time, being used to extract such an attack. Further, since the cache system is the most complex holder of speculatively updated soft state, it is responsible for the vast majority of overheads in mitigation. We argue here that for other system components, the overhead of providing Strictness Order is likely to be negligible, and that we can reuse techniques developed here or elsewhere.

Within-core structural hazards Strictness Order can be violated by a resource anywhere within the core being taken by a misspeculated instruction, delaying the scheduling of an instruction that will be committed. For single-cycle or pipelined operations, the solution is simple: we can prioritise instructions with lower timestamp. For multi-cycle operations that can cause structural hazards [5, 10, 39] scheduling can be done in Strictness Order via TimeGuarding. Alternatively, scheduling could be done optimistically, with leapfrogging used to cancel operations if Strictness Order is violated. We implemented the former in our gem5 simulation, such that functional units that are not pipelined (in our case, the IntDiv, FloatDiv, and FloatSqrt units) may only be issued a speculative operation once all previous speculative operations in timestamp order, that may use the same unit, have issued. This means that all IntDivs, for example, are executed in (speculative) program order with respect to each other. This causes no non-negligible slowdown in any workload we evaluated (maximum 0.08%). In fact, we saw a slight speedup on some workloads (4% geomean on SPEC CPU2006, and 2% on SPECspeed 2017). This is unsurprising; non-pipelined functional units are rarely part of address calculation for loads, and so are unlikely to be improved by long-scale reordering, and by favoring the scheduling of earlier operations, we cause entries to leave the reorder buffer more quickly. Since this shows a speedup, not a slowdown, we do not include this cost saving in the rest of the evaluation, to avoid masking overheads of other mitigations.

Load-Store Queue Here, protection is simpler than for the cache; it is naturally designed to transmit data in forwards-program order, thus providing Temporal Order for data flow. Still, to protect against the opposite (data not flowing when it otherwise should, via contention), TimeGuarding (section 4.4) and leapfrogging (section 4.5) can be used.

Transient contention from multiple threads Because no total ordering exists for concurrently executing instructions on multiple threads, leapfrogging cannot be used to hide contention between them, particularly on MSHR resources in shared caches. Data-oblivious execution [39] presents one solution to this, where each instruction has a predicted use of such resources, and must repeat once non-speculative if this prediction is incorrect. Still, the concept of Strictness Order makes this problem much simpler. As it provides a method of avoiding timing leakage in the wrong order within a thread, it vastly simplifies the prediction complexity. Instead of needing to predict *for every instruction*, Strictness Order makes predicting resource utilisation *per thread* sufficient to avoid leakage. This allows simple macro-level allocation based on recent committed behaviour to be used for shared resources. For simultaneous multi-threaded systems, cross-thread instruction port contention [6] can be prevented similarly.

Address translation GhostMinions should also be attached to TLBs [1] and page table walker caches. Behaviour is similar to those developed here, without coherence protection.

DRAM contention Speculation-hiding techniques in the

literature disallow speculative DRAM accesses entirely [39] or keep them out of scope [1, 38]. The biggest issue is open-page policies acting as an implicit cache. Still, allowing only non-speculative accesses to leave pages open may be feasible, since they are most useful with the spatially local accesses from prefetchers rather than the out-of-order core.

Other soft state: There are many structures that store soft state within systems, such as branch predictors, branch target buffers, and even frequency-voltage scaling [31]. Unlike with the cache, we believe there is little need to update these with speculative values, and so Strictness Order should in general be achieved by only updating them non-speculatively.

4.10 Optimizations

This proof-of-concept microarchitecture, implementing Temporal Order, leaves potential optimisations on the table.

Early Commit Instead of treating instructions as speculative until commit (as in GhostMinion, MuonTrap [1], InvisiSpec-Future [38] and STT-Future [40]), instructions could be treated as non-speculative provided their branches are resolved (as in InvisiSpec-Spectre [38] and STT-Spectre [40]). This requires some more tracking within the processor, and stops any inherent protection against exception attacks [21], but these may be dealt with via other means [32].

Full Strictness Order The Temporal Order that GhostMinion implements is, though simple, more restrictive than necessary. To support Strictness Order, rather than labelling each individual instruction with a new timestamp, this could instead be done by assigning a new timestamp after each speculatively predicted branch instruction (or more generally, operation that has the potential to fault). To extract further performance, groups with differing timestamps could be merged when intermediate speculation is resolved.

4.11 Summary

Here we have presented GhostMinion, a cache system designed to obey Temporal Order, a simple version of Strictness Order. To avoid any information leakage to concurrent instructions that violate this order, we have introduced a variety of techniques to make such instructions' behaviour visible when in a permitted order, and invisible otherwise: this is greatly simplified by the fact that Strictness Order is total for a pipeline's execution, and thus for any two instructions, at least one can see the effects of the other.

5 Experimental Setup

We use gem5 [8] to model the high-performance Aarch64 system in table 1. This is based on the default out-of-order setup in gem5 also used in previous Spectre defenses [1, 38, 40]. We simulate SPEC CPU2006 [14] in syscall emulation mode, fast forwarding for 1 billion instructions, then running for 1 billion. SPECspeed 2017 is fast-forwarded for 10 billion, to account for longer initialization, for workloads that would run in gem5. Parsec [7] (sim-medium) is run in full-system mode on four threads, to completion, for workloads that compile on Aarch64. Performance breakdowns are performed on SPEC CPU2006, as this shows the highest overhead. Results for InvisiSpec [38], STT [40] and MuonTrap [1], on the same Aarch64 setup, are compared against.

Core	
Core	4-Core, 8-Wide, Out-of-order, 2.0GHz
Pipeline	192-Entry ROB, 64-entry IQ, 32-entry LQ, 32-entry SQ, 256 Int / 256 FP registers, 6 Int ALUs, 4 FP ALUs, 2 Mult/Div ALU
Tournament Predictor	2-bit, 2048-entry local, 8192 global, 8192 choice, 4096 BTB, 16 RAS
Caches with GhostMinions	
L1 ICache	32KiB, 2-way, 2-cycle latency, 4 MSHRs
L1 DCache	64KiB, 2-way, 2-cycle latency, 4 MSHRs
D/I GhostMinions	2KiB, 2-way, accessed with I/D cache
Rest of system	
L2 Cache	2MiB, shared, 8-way, 20-cycle latency, 20 MSHRs, stride prefetcher (64-entry RPT)
Memory	DDR3-1600 11-11-11
OS	Ubuntu 14.04 LTS

Table 1: System experimental setup.

6 Evaluation

The performance overheads of GhostMinion are minimal for most workloads (2.5% geomean on SPEC CPU2006, 0% Parsec, 0.6% SPECspeed 2017). Though the worst case is 30%, this is a fundamental to hiding incorrect speculation rather than specific to GhostMinion. The prevention of backwards-in-time channels, via TimeGuarding and leapfrogging, rarely affects performance, as such flow is rare.

6.1 Comparison

Figure 6 shows the performance of GhostMinion on SPEC CPU2006 relative to other techniques in the literature. Figure 7 shows the same on 4-thread Parsec, and Figure 8 on SPECspeed 2017. While InvisiSpec [38] follows a similar strategy of hiding speculative loads (in a Load-Store Queue rather than a cache, and without precise Strictness Order guarantees, making it vulnerable to Speculative Interference [5]), its overheads are much higher: this is mainly down to the coherence protocol requiring a repeated load when an instruction becomes non-speculative before it is allowed to commit, unlike GhostMinion's MuonTrap-like technique [1] which takes such operations off of the critical path. This is despite GhostMinion allowing the use of set-associative structures in a leak-free way via TimeGuarding, and without redundant copies of a cache line for every load-queue entry. GhostMinion follows the same threat model as InvisiSpec-Future rather than InvisiSpec-Spectre, in that all operations are considered unsafe until commit time, rather than until all branches have been decided (but not exceptions). This has little impact on the performance of GhostMinion, because few critical-path operations must wait until commit.

The speculation-hiding mechanism of GhostMinion, made safe using the Strictness Order-based TimeGuarding and leapfrogging, allows more permissive execution of programs than the speculation-prevention mechanisms within STT [40], which only allow potentially-leaking memory ac-

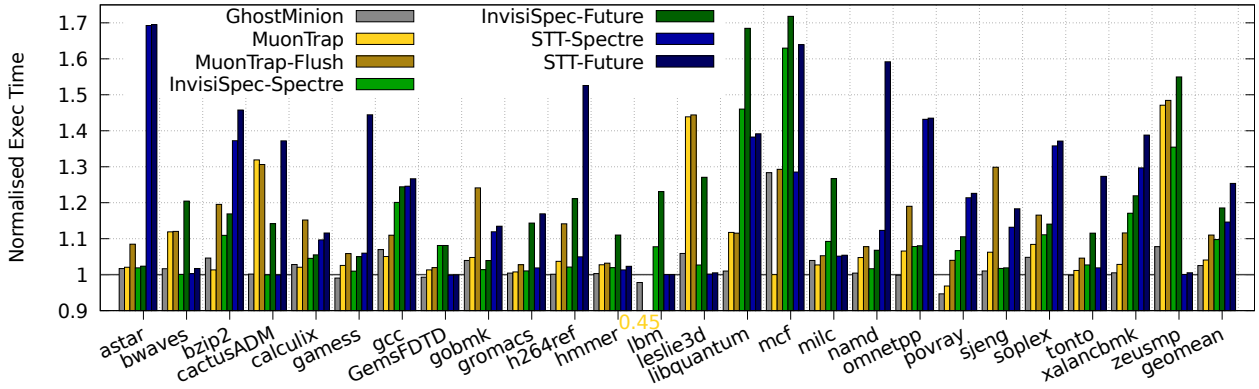


Figure 6: The performance overhead of GhostMinion on SPEC CPU2006, versus various techniques from the literature.

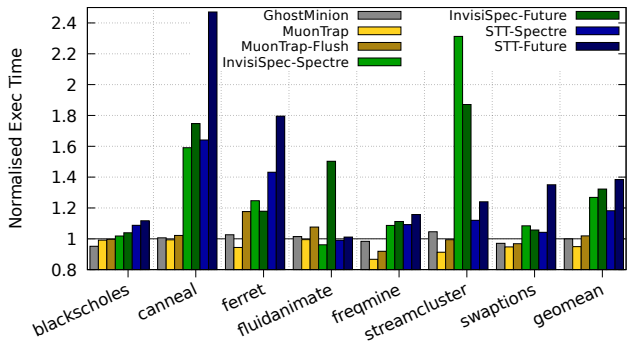


Figure 7: The performance overhead of GhostMinion on 4-thread Parsec, versus various techniques from the literature.

cesses, such as those based on another speculative (tainted) load, to occur once all previous branches have been decided. There are many workloads, such as astar, gamess, gromacs, libquantum, namd, omnetpp and xalancbmk, where STT shows large overheads when GhostMinion shows none at all.

Despite offering stronger guarantees than MuonTrap [1], a technique that hides speculative memory accesses in an L0 filter cache, rather than a parallel structure next to the L1, GhostMinion outperforms both MuonTrap techniques on SPEC CPU2006, and outperforms MuonTrap-Flush on Parsec and SPECspeed 2017. MuonTrap is designed primarily as a cross-process defence, though with an optional flush post-misspeculation (MuonTrap-Flush). Even though one of GhostMinion's mechanisms to support Strictness Order is this same flush, the placement of GhostMinion next to the L1 cache, with access in parallel, is better suited to frequent flushing than an L0 filter cache is, where the latter is often empty, and thus serves only to increase the access latency of the L1 data/instruction caches. Since a GhostMinion does not store values already in the attached L1, it can be smaller and less associative than a MuonTrap.

Some workloads still show significant overhead with a GhostMinion. Mcf in SPEC CPU2006 shows almost 30% overhead. MuonTrap's base system shows no overhead here, whereas STT-Spectre and MuonTrap-Flush show similar overhead to GhostMinion, with other techniques higher. The base MuonTrap is the only technique that allows access to data brought in via misspeculation past the point of the processor restarting execution: MuonTrap-Flush and GhostMinion clear it, InvisiSpec makes it invisible through its Load-Store queue, and STT doesn't execute dangerous loads at all. What this suggests is that mcf, and other workloads such as bzip2, gcc, gobmk, soplex, wrf and zeusmp, rely on this incorrect speculation to do useful prefetching work, and so any safe mechanism must prevent it and lose the relevant performance gains. To improve this, we would require more accurate speculation, or a (speculation-safe) prefetcher able to target complex memory accesses.

6.2 Analysis of Overheads

Figure 9 breaks down the overheads of each part of the full GhostMinion on SPEC CPU2006. Here, we see that most of the performance overhead comes from both the data-side GhostMinion, and the coherence protocol extensions, with a smaller contribution from the prefetcher (helping lbm by presenting a stabler pattern, and hindering others by affecting timeliness) and none from the instruction side.

Gcc, leslie3d and mcf are hurt by the lack of access to misspeculated data on the data-side, as is wrf in SPECspeed 2017. Still, a DMinion improves performance slightly on workloads such as gamess, GemsFDTD, tonto, povray, and particularly libquantum. This is down to reduced conflict misses, particularly when part of the active set is used repeatedly in the DMinion before it is committed (and thus evicts the conflicting sets). Still, this rare speedup is brittle; we see for Libquantum that, if either the coherence or prefetcher mechanisms are added, the improvement disappears.

6.3 TimeGuarding and Leapfrogging

The overheads from protecting against backwards-in-time channels within current speculative state, as opposed to lack of resource in the GhostMinion or clearing on misspeculation, rarely affect benign code. As seen in fig. 9, the extra overhead is 0.2%, with a worst case of 2.5%. Figure 10 shows the proportion of memory requests that experience *TimeGuarding* in the GhostMinion, to prevent access to a cache line brought in by a future memory request, those that experience *timeleaping*, where a memory request currently in the MSHRs would deliver data backwards in time, and *leapfrogging*, where limited miss status handling registers (MSHR) resources are taken in the wrong order, and therefore must be reclaimed by instructions with earlier times-

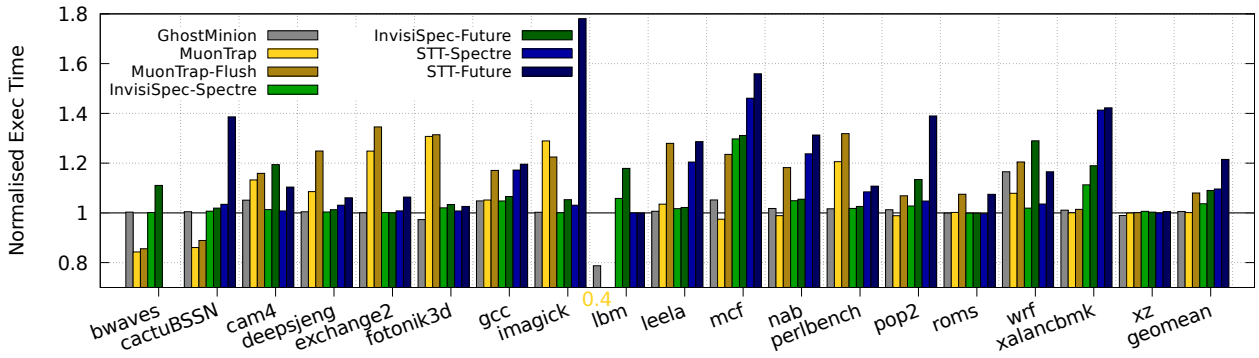


Figure 8: Performance on SPECspeed 2017.

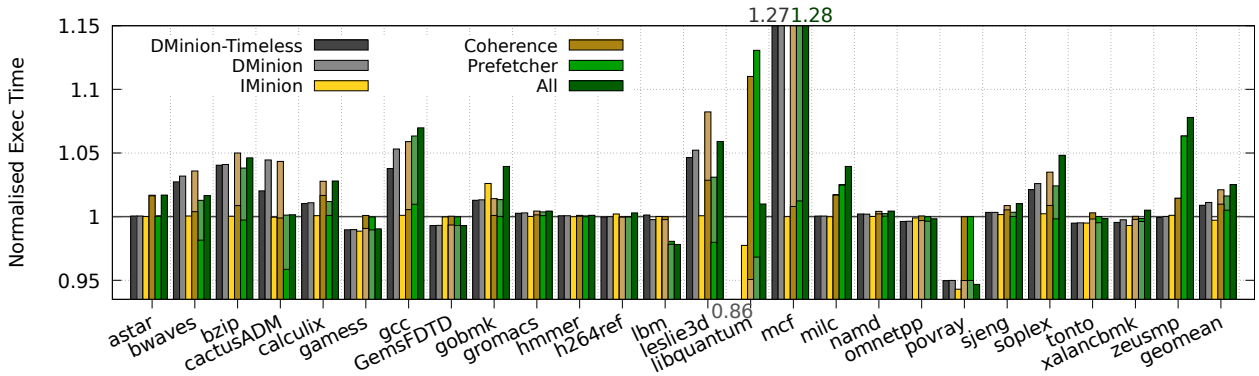


Figure 9: Breakdown of overheads from GhostMinion. Dminion-Timeless shows a GhostMinion with no concept of timestamping, but still wiped on misspeculation, so able to protect against standard Spectre attacks but not backwards-in-time attacks. The DMinion, IMinion and combined techniques are relative to an unprotected system; for Coherence / Prefetcher, the lighter bars show overhead (with DMinion enabled) relative to the baseline, and the darker bars relative to the DMinion alone.

tamps.

All three are uncommon; programs that transmit data "backwards in time" are unusual. Still, there are exceptions. Soplex often has *timeleaps*, or load hits in the MSHRs, which are waiting for memory accesses to complete, where the original instruction that caused the MSHR to be allocated is logically after the new access to the cache in timestamp order, causing a replay of the load to ensure safety. This may be limited by more permissive Strictness-Order mechanisms than the Temporal Order implemented in our prototype. Other workloads, such as mcf, libquantum, and omnetpp, experience *leapfrogging*, caused by a combination of running out of MSHR resources, and MSHR resources being allocated in an order where later timestamped instructions take resources that would otherwise be allocated to those with an earlier timestamp. Still, the overheads could be reduced by providing more MSHRs in the cache.

6.4 Sizing Sensitivity

Figure 11 shows the performance of GhostMinion with various sizes of GhostMinion, where 2KiB is the default. We see that 4KiB is negligibly faster and 1KiB is negligibly slower. For some workloads, performance is relatively consistent regardless of GhostMinion size; most of their working set fits in the L1, and so use of the GhostMinion is rare in general. Others (e.g. leslie3d) feature a steady slowdown as the lack

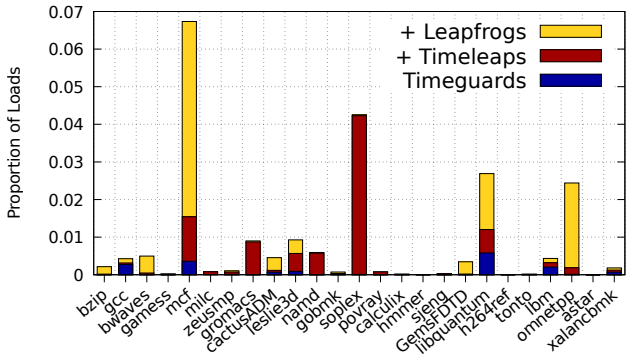


Figure 10: *Backwards-in-time* prevention is rarely triggered.

of space causes progressively more cache lines to be thrown away before reaching the non-speculative caches.

Still, from 512B (8 cache lines) or fewer, spikes start to appear: in particular, povray at 128B, and Xalancbmk at 512B and 256B (but not 128B). In these cases, the GhostMinion is consistently not big enough, and so cache lines leave the GhostMinion before commit. This means they are repeatedly brought in from main memory. This can be solved by asynchronously reloading cache lines that were brought in to the GhostMinion by a speculative instruction, but are missing from the L1 cache and GhostMinion at commit

time. This eliminates the spikes, and thus allows reasonable performance even with very small (128B, or 2 cache line) GhostMinions. For more moderate sizes, this lowers performance, as it causes reload of some values that are never used again, especially in GemsFDTD and to a lesser extent in zeusmp, milc and lbm, so we turn it off by default.

6.5 Power Analysis

GhostMinion involves accessing the L1 cache and GhostMinion in parallel, unlike MuonTrap [1] where the speculative-hiding cache is an L0 filter cache. We have shown in section 6.1 that this gives performance improvements when the speculative cache is frequently cleared, as is necessary for a comprehensive protection mechanism, but clearly there is an associated energy cost from the parallel accesses. Still, the timestamps that allow GhostMinion to safely use set associativity for its secret structures give significant efficiency compared with techniques that rely on full associativity or overprovisioning [16, 38] to hide storage contention.

At 22nm, CACTI [23] places the static power consumption at $0.47mW$ per 2KiB GhostMinion, versus $12.8mW$ for the 64KiB L1 data cache, and read energy at $1.5pJ$ versus $8.6pJ$ for the L1 data cache, due to the latter's larger size. The new memory accesses from GhostMinion constitute of a read access in the GhostMinion for every read in the L1, a write for every line brought into the GhostMinion, and a read for every cache line brought into the GhostMinion and then committed into the L1. While some of this could be reduced by a form of way prediction to only access one of the two structures, provided this maintained the Strictness Order guarantees, the overhead is already small, with $3\mu W$ maximum dynamic power draw for data, and $1\mu W$ for instructions, across SPEC CPU2006. Compared with around $1W$ per core for recent Arm systems [25], the overheads are negligible.

7 Related Work

Here we discuss existing (micro)architectural mitigations designed to prevent Spectre attacks. In turn, we consider how the concept of Strictness Order relates to their operation.

7.1 Speculation Hiding

Speculation-hiding techniques are intended to allow speculation to occur, but hide any effects from being observed by an attacker. There is varying compliance with Strictness Order, and thus vulnerability to contention attacks [5, 10].

InvisiSpec [38] appears to offer Temporal Order (definition 2) for loads only: "*InvisiSpec reuses only state allocated by prior (in program order) instructions, to avoid creating new side channels.*" To avoid incorrect eviction of data from its structures, it allocates them in program order. This requires overprovisioning, with full cache-line copies per each word-sized load-queue entry. SafeSpec [16] appears to offer Strictness Order (definition 1): "*Speculative instructions in the same execution branch as the load that fetched a shadow cache line... can use the value from the shadow structure.*" It worst-case overprovisions its shadow structures to avoid contention from concurrent execution. Unlike InvisiSpec and GhostMinion, coherence channels [34] are not protected, and only GhostMinion handles MSHR and within-core [5, 10] contention. Temporal Order directly translates into TimeGuarding in GhostMinion. This allows multiple levels of untrusted speculation to coexist within a set-associative cache structure, without leaking data via presence or absence. It gives more comprehensive enforcement, and without the overprovisioning of previous work [16, 38].

Data-oblivious execution [39] makes all speculative instructions use a predicted amount of time on resources such as cache (by level prediction) and FPU (estimated cycle length). Under Strictness Order, many side channels are permitted without obliviousness: earlier instructions in program order always allow Strictness-Order transmission to later ones.

DAWG [17] dynamically partitions cache ways between distinct domains, such as groups of processes; MuonTrap [1] prevents the observation of speculative state, brought in to a filter cache, from affecting other processes, by flushing limited regions of speculation on context switch, and via coherence protocol changes. Neither enforces Strictness Order even between processes: though a cross-process attacker may not be able to directly observe speculative state changes, the timing of context switches and inter-process communication may still be altered by the effects of misspeculated execution. GhostMinion's TimeGuarding provides a direct method to augment these caches to support hiding concurrent execution, by preventing information flow backwards-in-time.

Rollback techniques, such as CleanupSpec [27] and ReversiSpec [37], permit speculative execution to change the cache system, and roll it back under misspeculation. These violate Strictness Order on two counts. The time to roll back is dependent on the amount of state, which affects restarting of execution. As exploited in SpectreRewind [10] and Speculative Interference [5], correct execution, earlier in program order but concurrent with misspeculated execution, can see the effects of misspeculation. The latter property affects any techniques that only clear effects of misspeculation once it is discovered, such as MuonTrap-Flush [1], Ghost Loads [28], and Gonzalez et al.'s RISC-V BOOM mitigation [11].

Spectector [12] is a software analysis technique, based on the concept of *speculative non-interference* of state leakage. DOLMA [22] defines a similar property in *Transient Non-Observability*. He et al. [13] present the concept of attack graphs, which model access-control authorization delays on secret data in order to generate new vulnerabilities and evaluate defences against them. Strictness Order is a permissive constraint that can be used to define hardware that comprehensively obeys and defends against these properties.

7.2 Speculation Restricting

Alternatively, techniques can restrict speculative behaviour from occurring at all, rather than hiding its effects. Examples of such techniques include Speculative Taint Tracking [40], NDA [36], Selective Delay and Value Prediction [29], Conditional Speculation [20], and SpecShield [4]. Restricting unsafe speculation is sufficient to achieve Strictness Order, though it is not necessary. Still, restricting it entirely has fewer implications on complexity, state-hiding and coherence. To limit performance impact, such techniques violate Strictness Order in a selective way, for speculative instructions that can be demonstrated unlikely to leak secrets, for example not taking input from speculatively loaded data [40].

The fence instructions [2, 15, 33] built in to current mi-

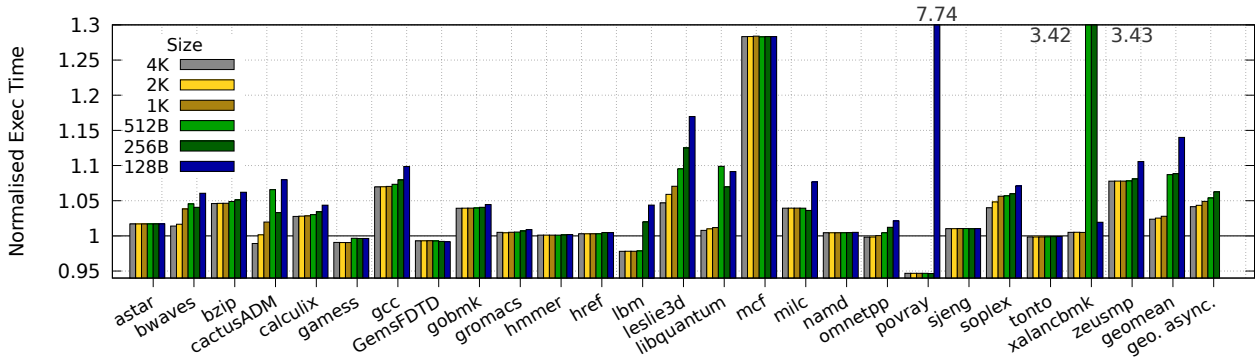


Figure 11: Slowdown of GhostMinion with different-sized instruction and data minions (2K default, each 2-way assoc.). Asynchronous reload of lost cache lines (*geo. async.*, where we only show the geomean for brevity) removes the spikes in performance seen with small GhostMinions, but is inadvisable for systems with ample resources.

croarchitectures can be viewed as a special case: these limit reordering of instructions around each side of the fence. Still, these could be implemented, without changing their guarantees, via Strictness Order around the barrier instead.

7.3 Other Techniques

Other more ad-hoc speculation-restriction techniques are currently in use, such as Retpoline [35], which redirects branch target flow, though these are harder to give precise Strictness Order analysis of. Isolation and randomisation of structures, such as over the branch target buffer [3] and branch predictor [19], restrict the ability of a cross-domain attacker from training a victim. Though they do not prevent attacks where the victim trains itself, and are thus more suited to protection against Variant 2 [18] than other attacks, they provide strong probabilistic guarantees rather than full Strictness Order constraints. Kaiser [32] is another example of isolation, where Meltdown [21] is prevented by separating out the kernel and userspace's page tables, and thus gives protection outside of Strictness-Order guarantees.

8 Conclusion

We have presented Strictness Ordering, a permissive constraint system designed to prevent speculative execution attacks, by comprehensively preventing the timing effects of mis-speculated instructions from leaking to instructions that do commit, either executed concurrently or in future. It does this while still allowing widespread forwarding of speculative information, by noting that, within a thread, the fates of many speculative instructions are tied together, and that processors generate instructions that become progressively more speculative, and so instructions earlier in program order can always produce side channels for those later.

GhostMinion demonstrates the techniques needed to implement a full microarchitecture without heavy speculation restriction, while remaining safe via Strictness-Order guarantees. By combining Strictness Order and GhostMinion with prediction for shared speculative resources between threads [39], and notions of Speculative Taint [40] to further permissify constraints, we believe systems will finally be able to rigorously eliminate transient-execution attacks at acceptable costs, ending this cat-and-mouse game for good.

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant reference EP/V038699/1. Additional data related to this publication is available at `https://github.com/SamAinsworth/reproduce-ghostminion-paper` or `https://doi.org/10.5281/zenodo.5222208`.

References

[1] S. Ainsworth and T. M. Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *ISCA*, 2020.

[2] Arm, "Arm processor security update," `https://developer.arm.com/support/security-update/compiler-support-for-mitigations`, Jan. 2018.

[3] Arm, "Arm v8.5-a cpu updates," `https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/downloads/arm-v8-5-a-cpu-updates`, Feb. 2019.

[4] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "SpecShield: shielding speculative data from microarchitectural covert channels," in *PACT*, 2019.

[5] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, F. Mckeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. Alameldeen, "Speculative interference attacks: Breaking invisible speculation schemes," in *arXiv 2007.11818*, 2020.

[6] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: Exploiting speculative execution through port contention," in *CCS*, 2019.

[7] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, Aug. 2011.

[9] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *USENIX Security*, 2018.

[10] J. Fustos, M. Bechtel, and H. Yun, "SpectreRewind: Leaking secrets to past instructions," in *arXiv 2003.12208*, 2020.

[11] A. Gonzalez, B. Korpan, J. Zhao, E. Younis, and K. Asanović, "Replicating and mitigating Spectre attacks on an open source RISC-V microarchitecture," in *CARRV*, 2019.

[12] M. Guarnieri, B. Kopf, J. F. Morales, J. Reineke, and A. Sanchez, "Spectector: Principled detection of speculative information flows," in *SP*, 2020.

[13] Z. He, G. Hu, and R. Lee, "New models for understanding and reasoning about speculative execution attacks," in *HPCA*, 2021.

[14] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, Sep. 2006.

[15] Intel, "Intel analysis of speculative execution side channels," `https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf`, Jan. 2018.

[16] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "SafeSpec: Banishing the spectre of a meltdown with leakage-free speculation," in *DAC*, 2019.

[17] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *MICRO*, 2018.

[18] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *S&P*, 2019.

[19] J. Lee, Y. Ishii, and D. Sunwoo, "Securing branch predictors with two-level encryption," *ACM TACO*, vol. 17, no. 3, Aug. 2020.

[20] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *HPCA*, 2019.

[21] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018.

[22] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, "Dolma: Securing speculation with the principle of transient non-observability," 2020.

[23] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," in *MICRO*, 2007.

[24] A. Mycroft, "The theory and practice of transforming call-by-need into call-by-value," in *Proceedings of the Fourth 'Colloque International Sur La Programmation' on International Symposium on Programming*, 1980.

[25] G. Papadimitriou, A. Chatzidimitriou, and D. Gizopoulos, "Adaptive voltage/frequency scaling and core allocation for balanced energy and performance on multicore cpus," in *HPCA*, 2019.

[26] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *MICRO*, 1999.

[27] G. Saileshwar and M. K. Qureshi, "CleanupSpec: An "undo" approach to safe speculation," in *MICRO*, 2019.

[28] C. Sakalis, M. Alipour, A. Ros, A. Jimborean, S. Kaxiras, and M. Själander, "Ghost loads: What is the cost of invisible speculation?" in *CF*, 2019.

[29] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *ISCA*, 2019.

[30] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript," in *FC*, Jan. 2017.

[31] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *ESORICS*, 2019.

[32] J. Tang, "KAISER: Hiding the kernel from user space," `https://lwn.net/Articles/738975/`, 2017.

[33] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *ASPLOS*, 2019.

[34] C. Trippel, D. Lustig, and M. Martonosi, "Checkmate: Automated synthesis of hardware exploits and security litmus tests," in *MICRO*, 2018.

[35] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," `https://support.google.com/faqs/answer/7625886`, 2018.

[36] O. Weisse, I. Neal, K. Loughlin, T. Wenisch, and B. Kasikci, "NDA: Preventing speculative execution attacks at their source," in *MICRO*, 2019.

[37] Y. Wu and X. Qian, "Reversispec: Reversible coherence protocol for defending transient attacks," in *arXiv 2006.16535*, 2020.

[38] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making speculative execution invisible in the cache hierarchy," in *MICRO*, 2018.

[39] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution," in *ISCA*, 2020.

[40] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data," in *MICRO*, 2019.